Introduction to pin muxing and GPIO control under Linux

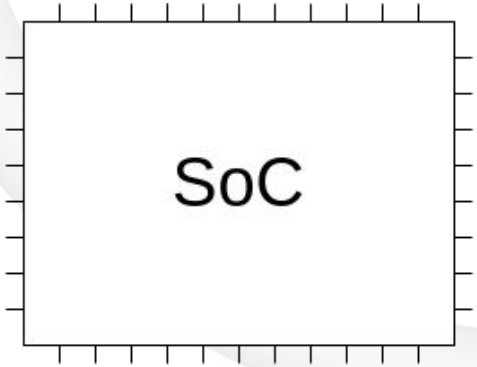
Neil Armstrong - BayLibre

#lfelc @superna9999

Hardware Level

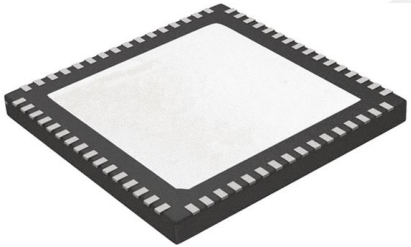
Hardware Level

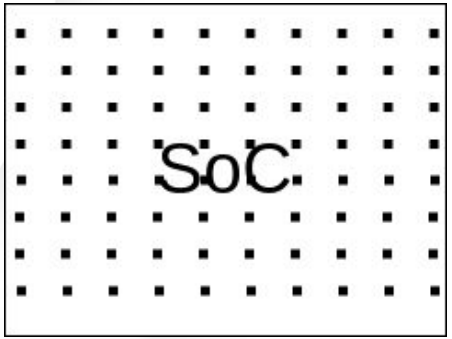
Pins or Balls on SoC Package



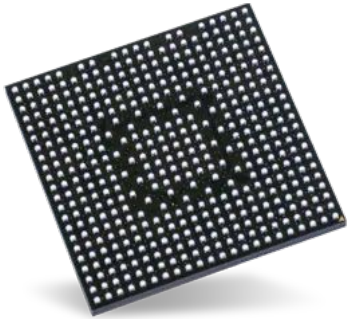
Quad Flat Package (QFP)

Quad Flat No-leads package (QFN)



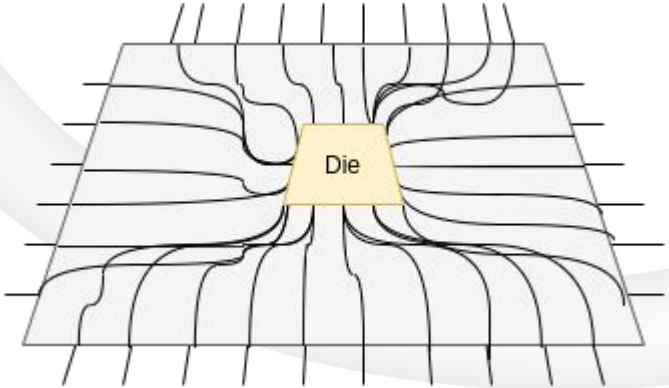


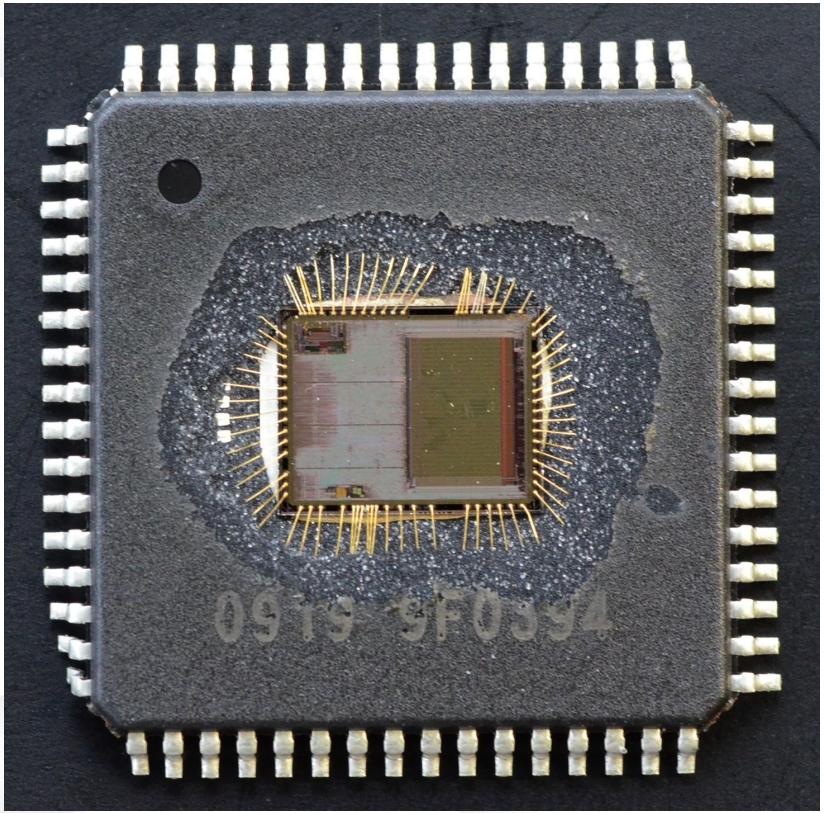
Ball Grid Array (BGA)



Hardware Level

Wires from the die to the external package pins or balls.





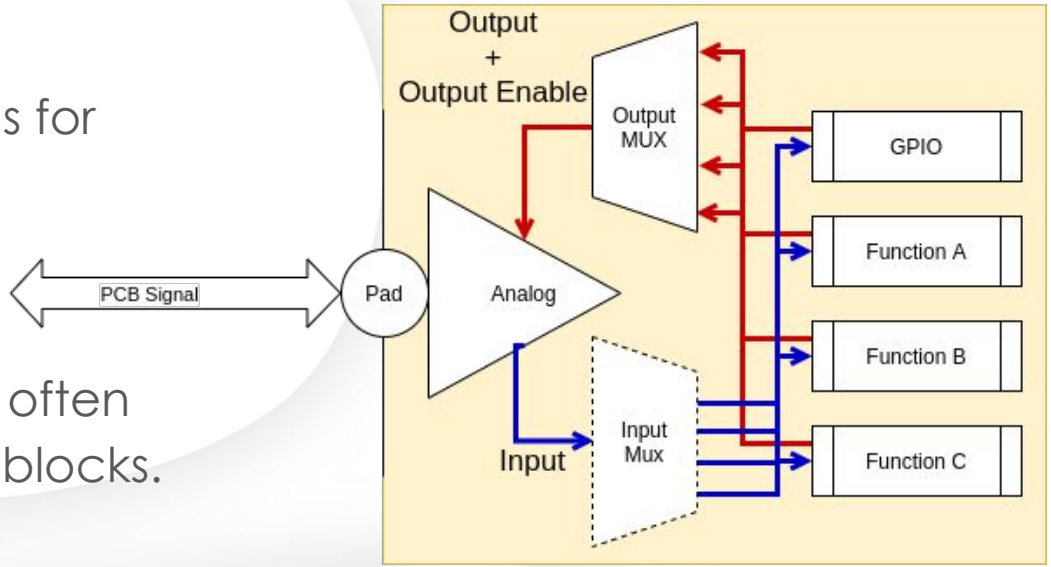
Hardware Level

Main HW block of a “PAD”

Can handle multiple functions for different internal HW blocks.

GPIO is one of the functions

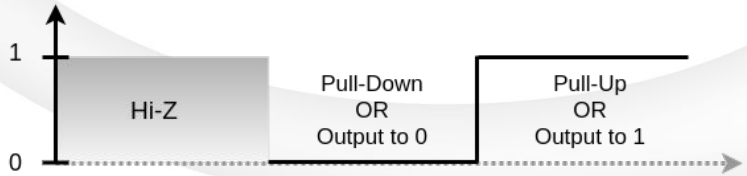
Input can be muxed of more often directly connected to all HW blocks.

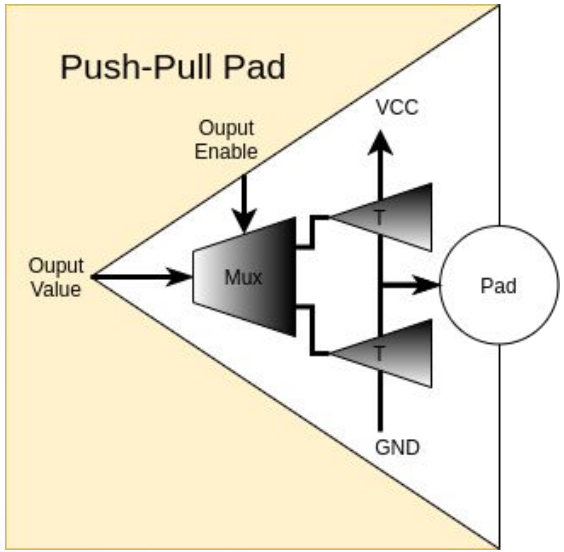


Hardware Level

SoC pads are generally “Tri-State”:

- Hi-Z: high impedance, not driven by the SoC
- 1 level: Driven to the pad voltage
- 0 level: Driven to ground

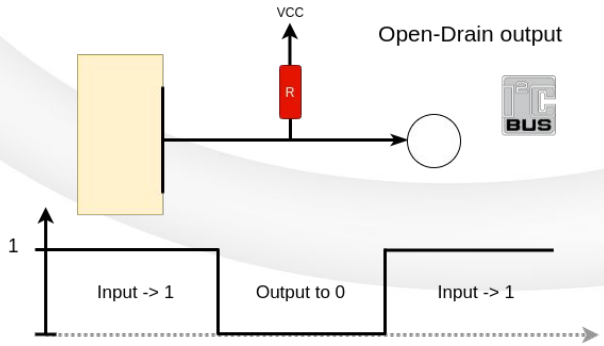


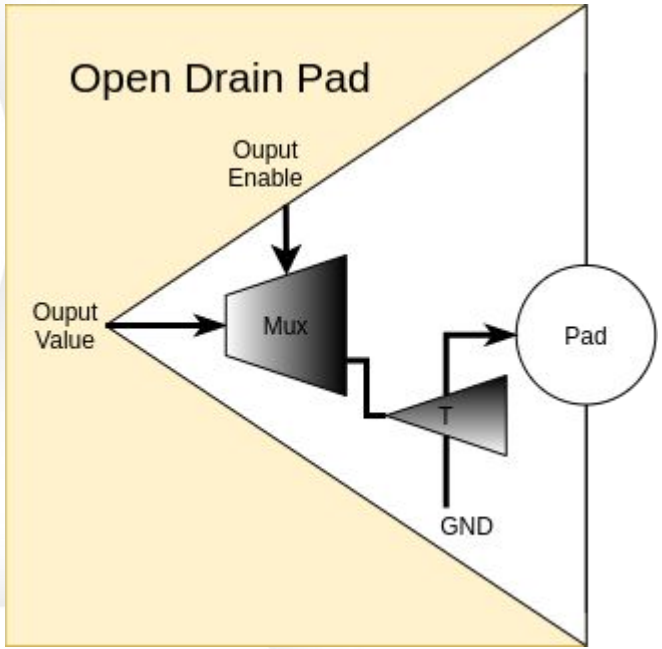


Hardware Level

An Open-Drain pad can only drive to 0 (ground)

Used for example for the I2C and 1-wire busses.

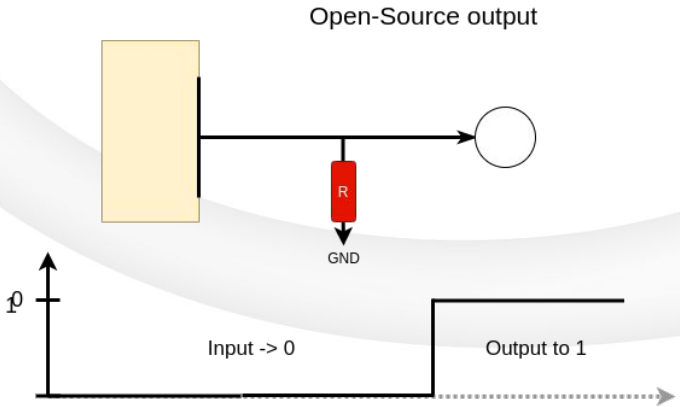


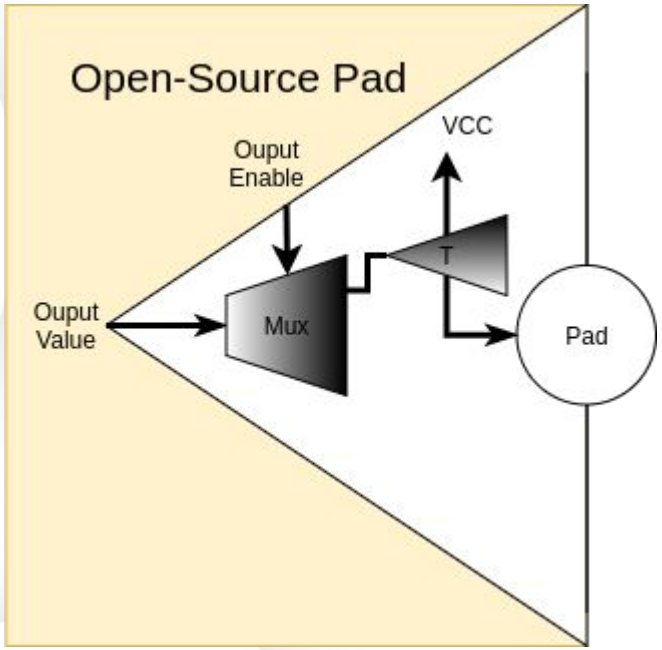


Hardware Level

An Open-Source pad can only drive to 1 (VCC).

Not widely used but exists.





Hardware Level

Analog pad HW also optionally have:

- Pull-Up: small internal resistor to VCC
- Pull-Down: small internal resistor to GND
- Drive strength: to select different output signal strengths
- ….

Hardware Level

PAD functions can be selected:

- Individually: most common => a number identifies the function
- By Group: groups multiple PADs for a specific function at once

Some weird setups exists:

- Multi-function PADs without GPIO
- GPIO-only PADs
- Function can be selected on multiple PADs (generally only output)

Software Level: Legacy

Software Level: Legacy

Before introduction of the PINCTRL framework:

ANARCHY !

Code was mainly in arch/arm/ and was from:

- Very basic hard coded pin setup for SoC and Board
  to
- Extremely complex framework with conflict checking

Software Level: Legacy

Lessons learned:

- Pin Conflict Checking is almost impossible to implement
  - Would need an insane level of code and HW description to handle
  - Is in fact useless, proper code review and testing is enough
- We need dynamic pin mux setup
  - Needed for low-power devices (Phones, Tablets, …)
  - Some Drivers needs pin mux / config change at runtime
- No more hardcoding
  - maintaining SoC/Board description C code is like hell
  - Not scalable

Software Level: Legacy

GPIO API was also ANARCHY.

API was not described in include/linux, but in each arch/arm/mach-XXXX/include/mach/gpio.h

Each SoC potentially had a different:

- API
- Implementation
- Behavior

Software Level: Legacy

- API was too basic
- Implementations were all different
- Multiple GPIO controllers was complex too handle
- Nested GPIO controllers was also impossible to handle
  - I2C, SPI, USB, … GPIO extenders
- Advanced API was only available on some SoC
- GPIO IRQ was also very SoC specific

Software Level: PINCTRL Framework

Software Level: PINCTRL Framework

The “pin control subsystem” submitted by Linus Walleij
was merged in Linux 3.2-rc2 in Oct 13, 2011

This creates a subsystem for handling of pin control devices.
These are devices that control different aspects of package pins.

Currently it handles **pinmuxing**, i.e. assigning electronic functions to groups of pins [...].
The plan is to also handle other **I/O pin control aspects** [...] to remove a lot of ARM arch code as well as feature-creepy GPIO drivers which are implementing the same thing over and over again.

Software Level: PINCTRL Framework

Definition of PIN CONTROLLER:

A pin controller is a piece of **hardware**, usually a set of registers, that **can control PINs**.

It may be able to **multiplex**, bias, set load capacitance, set drive strength etc **for individual pins or groups of pins**.

Software Level: PINCTRL Framework

Definition of PIN:

PINS are equal to **pads**, fingers, **balls** or whatever packaging input or output line you want to control and these are denoted by unsigned integers in the range 0..maxpin.

This **number space is local to each PIN CONTROLLER**, so there may be **several such number spaces in a system**.

This pin space may be sparse - i.e. there may be gaps in the space with numbers where no pin exists.

Software Level: PINCTRL Framework

Pin groups

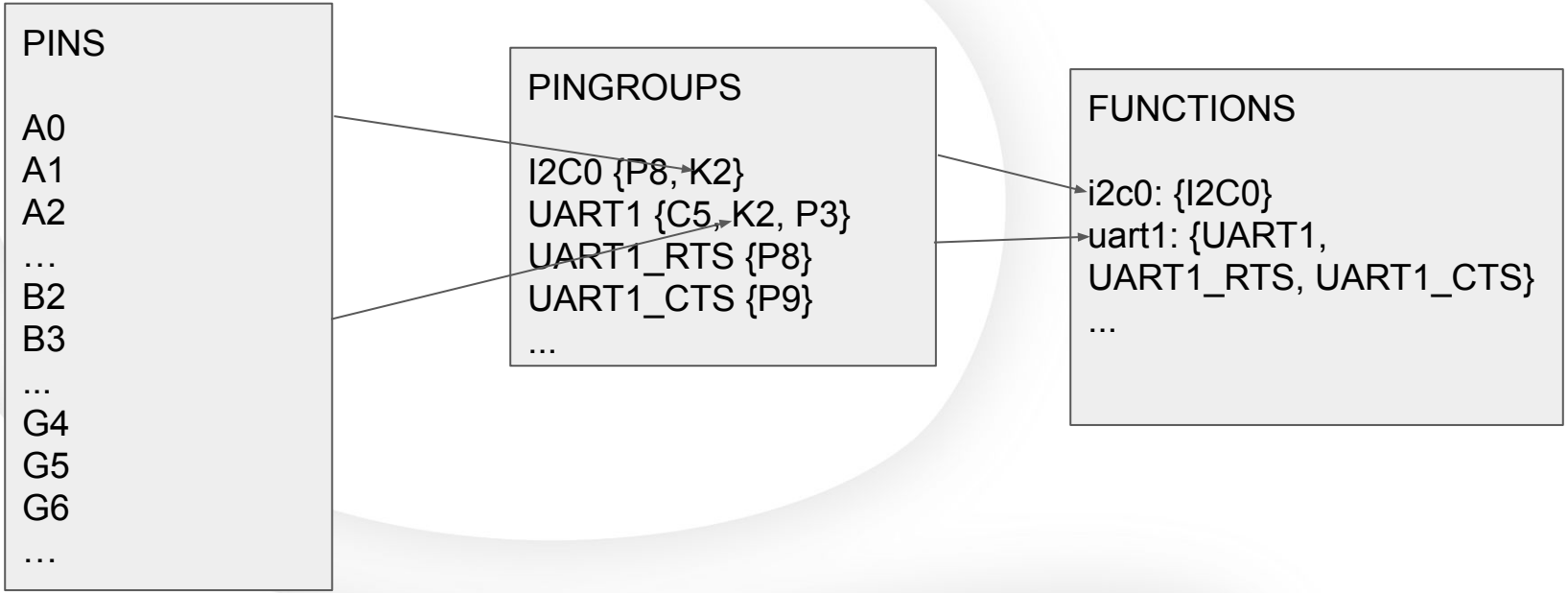
Many controllers need to deal with **groups of pins**, so the pin controller subsystem has a mechanism for enumerating groups of pins and retrieving the actual enumerated pins that are part of a certain group.

Software Level: PINCTRL Framework

PINMUX, also known as **padmux, ballmux, alternate functions or mission modes** is a way for chip vendors producing some kind of electrical packages to use a certain physical pin (ball, pad, finger, etc) **for multiple mutually exclusive functions**, depending on the application.

By "application" in this context we usually mean a way of soldering or wiring the package into an electronic system, even though the framework makes it possible to also **change the function at runtime**.

Software Level: PINCTRL Framework



Software Level: PINCTRL Framework

- Pins can have holes, and “pretty names” like Ball ID
  - The name reflects what the vendor uses to distinguish pins
  - But each must be unique
- Pingroups can have 1 to many pins
  - Multiple pingroups can share the same pins
- Functions associates a name to a set of pingroups
  - Multiple functions can share the same pingroups
  - But only one of these conflicting functions can be set at the same time

Software Level: PINCTRL Framework

- Pins <-> Groups <-> Function association is done in driver to match the underlying hardware
- Pinmux Framework needs :
  - Pins list

```
const struct pinctrl_pin_desc *pins;
unsigned int npins;
```

  - Pin control ops

```
const struct pinctrl_ops *pctlops;
```

  - Pin muxing ops

```
const struct pinmux_ops *pmxops;
```

  - Pin config ops

```
const struct pinconf_ops *confops;
```

Software Level: PINCTRL Framework

- `const struct pinctrl_ops *pctlops;`
  - `int (*get_groups_count) (struct pinctrl_dev *pctldev)`
    - Get the number of pin groups
  - `const char *(*get_group_name) (struct pinctrl_dev *pctldev, unsigned selector);`
    - Get the name (string identifier) of a pin group
  - `int (*get_group_pins) (struct pinctrl_dev *pctldev, unsigned selector, const unsigned **pins, unsigned *num_pins);`
    - Get the pins involved in a pin group

Software Level: PINCTRL Framework

- `struct pinmux_ops`
  - `int (*get_functions_count) (struct pinctrl_dev *pctldev);`
    - Get the number of functions
  - `const char *(*get_function_name) (struct pinctrl_dev *pctldev, unsigned selector);`
    - Get the name of a function
  - `int (*get_function_groups) (struct pinctrl_dev *pctldev, unsigned selector, const char * const **groups, unsigned *num_groups);`
    - Get the pin groups involved in a function
  - `int (*set_mux) (struct pinctrl_dev *pctldev, unsigned func_selector, unsigned group_selector);`
    - Set the pins of a groups in a function

Software Level: PINCTRL Framework

- Pinctrl subsystem was designed *before* Device Tree
- Pinmux mapping was hard-coded in arch/arm/mach-
- Now mapping is described in DT (or ACPI)
- Usage is still the same
  - `p = devm_pinctrl_get(dev);`
  - `s = pinctrl_lookup_state(p, PINCTRL_STATE_DEFAULT);`
  - `pinctrl_select_state(p, s);`

Software Level: PINCTRL Framework

- DT Mapping
  - ```
    uart1_pins: uart1 {
        mux {
            groups = "UART1", "UART1_RTS", "UART1_CTS";
            function = "uart1";
        };
    };
    ```
- From a device node
  - ```
    uart@123456 {
        pinctrl-0 = <&uart1_pins>;
        pinctrl-names = "default";
        …
    };
    ```

PINGROUPS

I2C0 {P8, K2}
UART1 {C5, K2, P3}
UART1_RTS {P8}
UART1_CTS {P9}
...

FUNCTIONS

i2c0: {I2C0}
uart1: {UART1, UART1_RTS, UART1_CTS}
...

Software Level: GPIO Framework

Software Level: GPIO Framework

The “gpio provider infrastructure” submitted by David Brownell
was merged in Linux 2.6.25-rc2 in Feb 5, 2008

Provide new implementation infrastructure that platforms may choose to use when implementing the GPIO programming interface.
Platforms can update their GPIO support to use this.

Finally an unified GPIO API and implementation.

Software Level: GPIO Framework

The “gpio provider infrastructure” is named “gpiolib”.

Each GPIO was identified as an unique number.

This number was used in data platform to identify GPIOs to use by drivers.

Handling of GPIO extender was possible, but this static numbering made it limited.

Software Level: GPIO Framework

The “gpio provider infrastructure” was progressively replaced by:

GPIOD

It replaced the “legacy” GPIO interface by maintaining the original API while introducing a new GPIO framework based around a **struct gpio_desc** for each GPIO for the consumers.

Internally, an unique number still identifies each GPIO.

Software Level: GPIO Framework

The legacy API is still usable from:

include/linux/gpio.h

But is removed bits per bits in favor or:

include/linux/gpio/ with **consumer.h** for drivers.

Software Level: GPIO Framework

Like the PINCTRL framework, the consumer and implementation API are separate.

For the GPIO controller implementation:

include/linux/gpio/driver.h

For the GPIO consumer:

include/linux/gpio/consumer.h

Software Level: GPIO Framework

include/linux/gpio/driver.h

Basic GPIO controller registration:

```
chip->gpio_chip.label = "NAME";
      direction_output() will be called to set input or output
chip->gpio_chip.direction_output = direction_output;
      get_value() will be called to get the GPIO input value (0 or 1)
chip->gpio_chip.get = get_value;
      set_value() will be called to set the GPIO output value (0 or 1)
chip->gpio_chip.set = set_value;
chip->gpio_chip.base = -1;
chip->gpio_chip.ngpio = NUMBER_GPIOS;
      parent links the GPIO controller to the device
chip->gpio_chip.parent = dev;

gpiochip_add_data(&chip->gpio_chip, chip);
```

Software Level: GPIO Framework

include/linux/gpio/consumer.h

With Device-Tree, gpios are linked to device nodes with:

```
audio-amplifier-0 {
        compatible = "simple-audio-amplifier";
        enable-gpios = <&gpio_ao GPIOAO_2 GPIO_ACTIVE_HIGH>;
        status = "okay";
};
```

Software Level: GPIO Framework

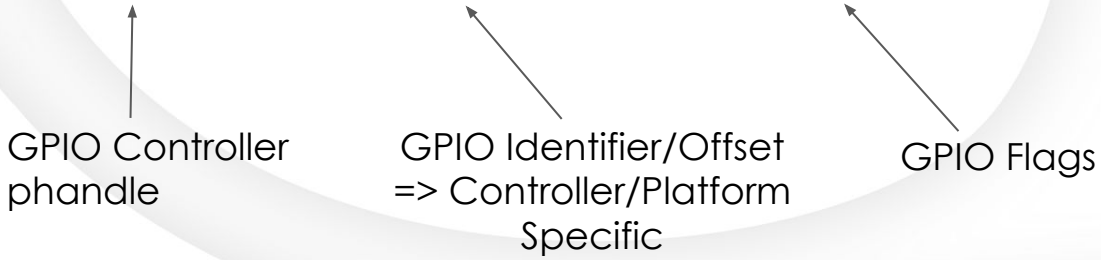
```
audio-amplifier-0 {
	compatible = "simple-audio-amplifier";
	enable-gpios = <&gpio_ao GPIOAO_2 GPIO_ACTIVE_HIGH>;
	status = "okay";
};
```

enable-*gpios*

“enable” is the “function” name associated to the GPIO, can be enything like “reset”, “shutdown”, ….

Software Level: GPIO Framework

```
audio-amplifier-0 {
        compatible = "simple-audio-amplifier";
        enable-gpios = <&gpio_ao GPIOAO_2 GPIO_ACTIVE_HIGH>;
        status = "okay";
};
```



Software Level: GPIO Framework

```
=> include/dt-bindings/gpio/gpio.h
/* Bit 0 express polarity */
#define GPIO_ACTIVE_HIGH 0
#define GPIO_ACTIVE_LOW 1

/* Bit 1 express single-endedness */
#define GPIO_PUSH_PULL 0
#define GPIO_SINGLE_ENDED 2

/* Bit 2 express Open drain or open source */
#define GPIO_LINE_OPEN_SOURCE 0
#define GPIO_LINE_OPEN_DRAIN 4

/* Bit 3 express GPIO suspend/resume and reset persistence */
#define GPIO_PERSISTENT 0
#define GPIO_TRANSITORY 8

/* Bit 4 express pull up */
#define GPIO_PULL_UP 16

/* Bit 5 express pull down */
#define GPIO_PULL_DOWN 32

/* Open Drain/Collector is the combination of single-ended open drain interface.
 * Open Source/Emitter is the combination of single-ended open source interface. */
#define GPIO_OPEN_DRAIN (GPIO_SINGLE_ENDED | GPIO_LINE_OPEN_DRAIN)
#define GPIO_OPEN_SOURCE (GPIO_SINGLE_ENDED | GPIO_LINE_OPEN_SOURCE)
```

Software Level: GPIO Framework

include/linux/gpio/consumer.h

Get the “enable” GPIO, and initialize it to “LOW” output:

```
gpiod_enable = devm_gpiod_get_optional(dev, "enable", GPIOD_OUT_LOW);
```

Set value:

```
gpiod_set_value(gpiod_enable, val);
```

Software Level: GPIO Framework

include/linux/gpio/consumer.h

With the DT flags (`GPIO_ACTIVE_HIGH` or `GPIO_ACTIVE_LOW`), the GPIO core will invert the GPIO value to match the "ACTIVE" polarity.

Active polarity is when the signal "acts"

- !RESET IC port will expect `GPIO_ACTIVE_LOW`
- EN IC port will expect `GPIO_ACTIVE_HIGH`

Software Level: GPIO Framework

include/linux/gpio/consumer.h

For example, if `GPIO_ACTIVE_LOW` is passed in DT flag:

- `gpiod_set_value(gpiod_enable, 0);` => will set the GPIO to 1
- `gpiod_set_value(gpiod_enable, 1);` => will set the GPIO to 0

Software Level: PINCTRL & GPIO

Software Level: PINCTRL & GPIO

- The PINCTRL framework is able to “link” itself to GPIO
- Often the GPIO controller is “inside”/”next” the PINCTRL HW
- Can specify “ranges” between a PIN controller and GPIO controllers

Thanks for listening !

Scan me for the slides